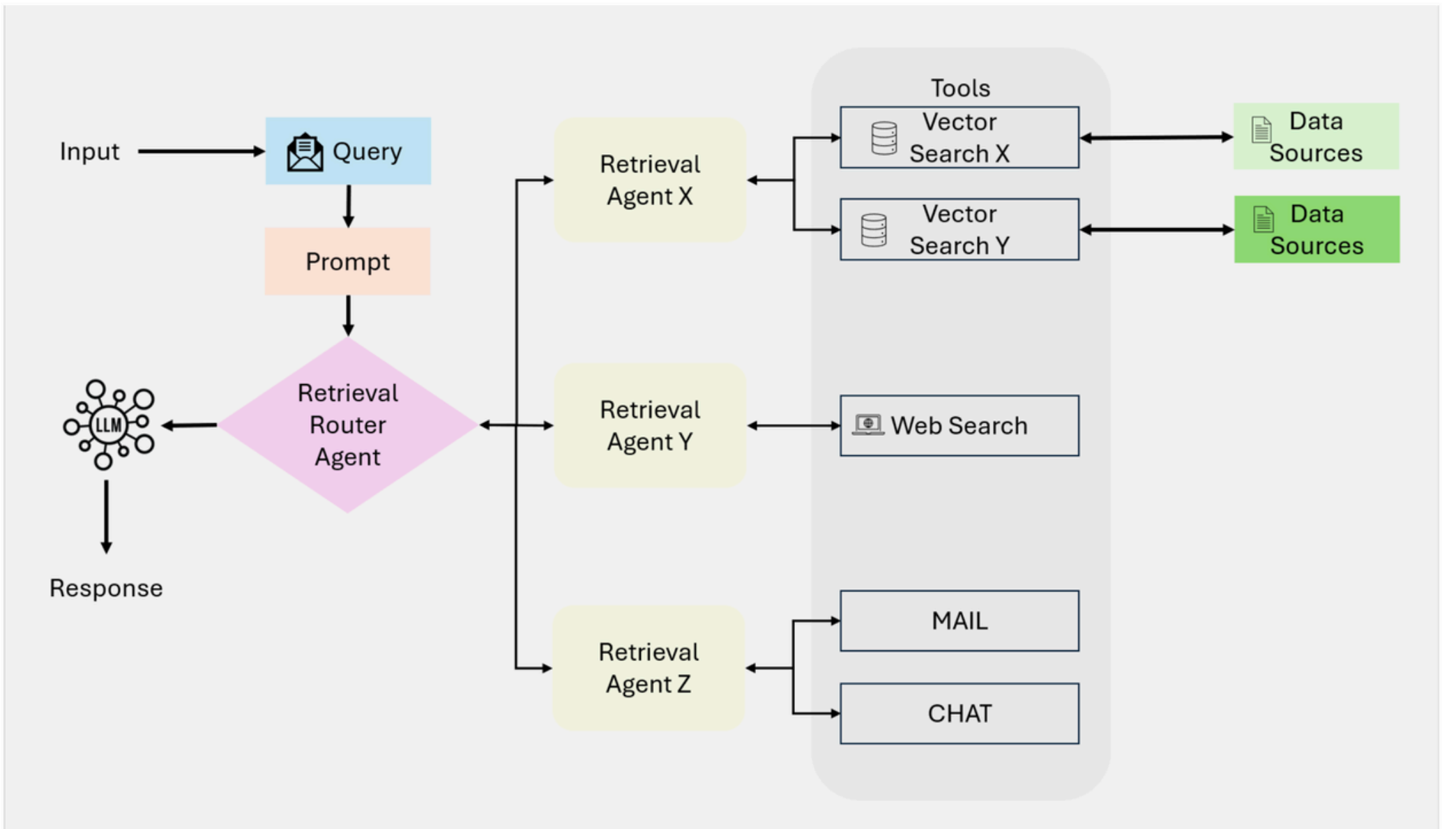
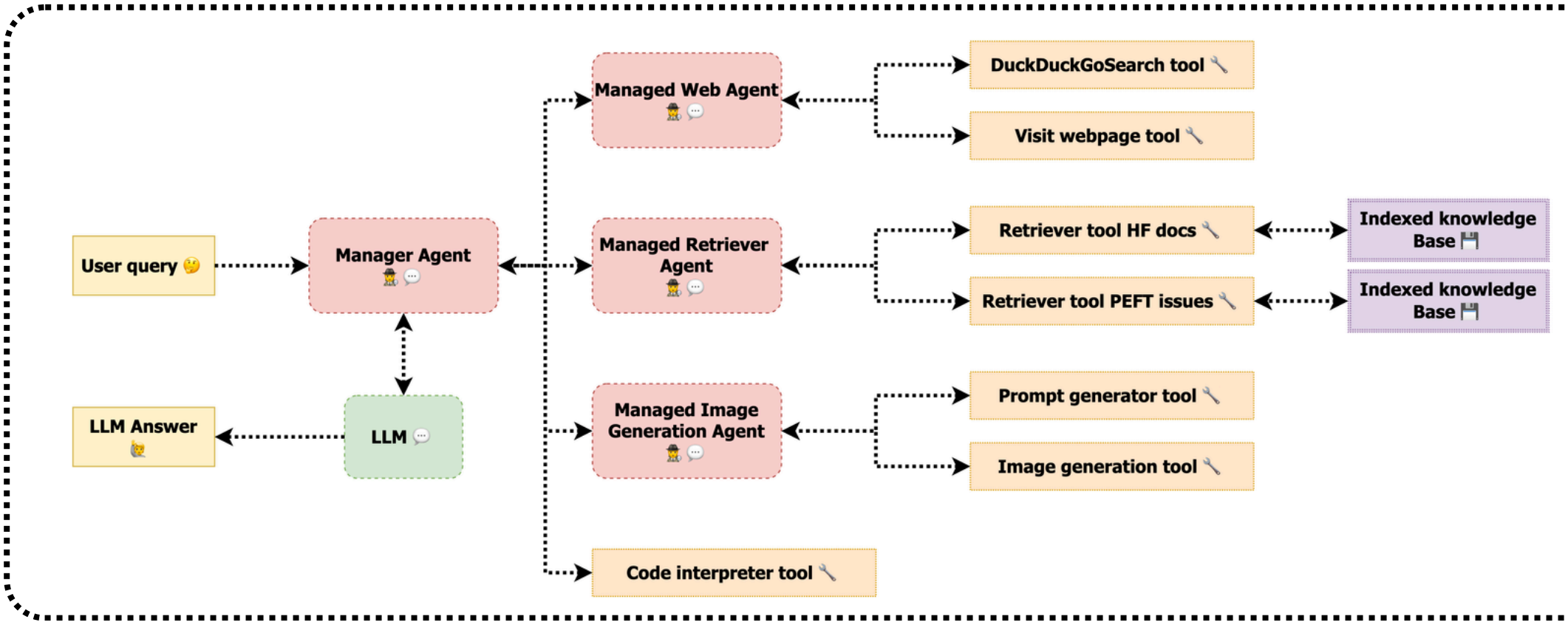
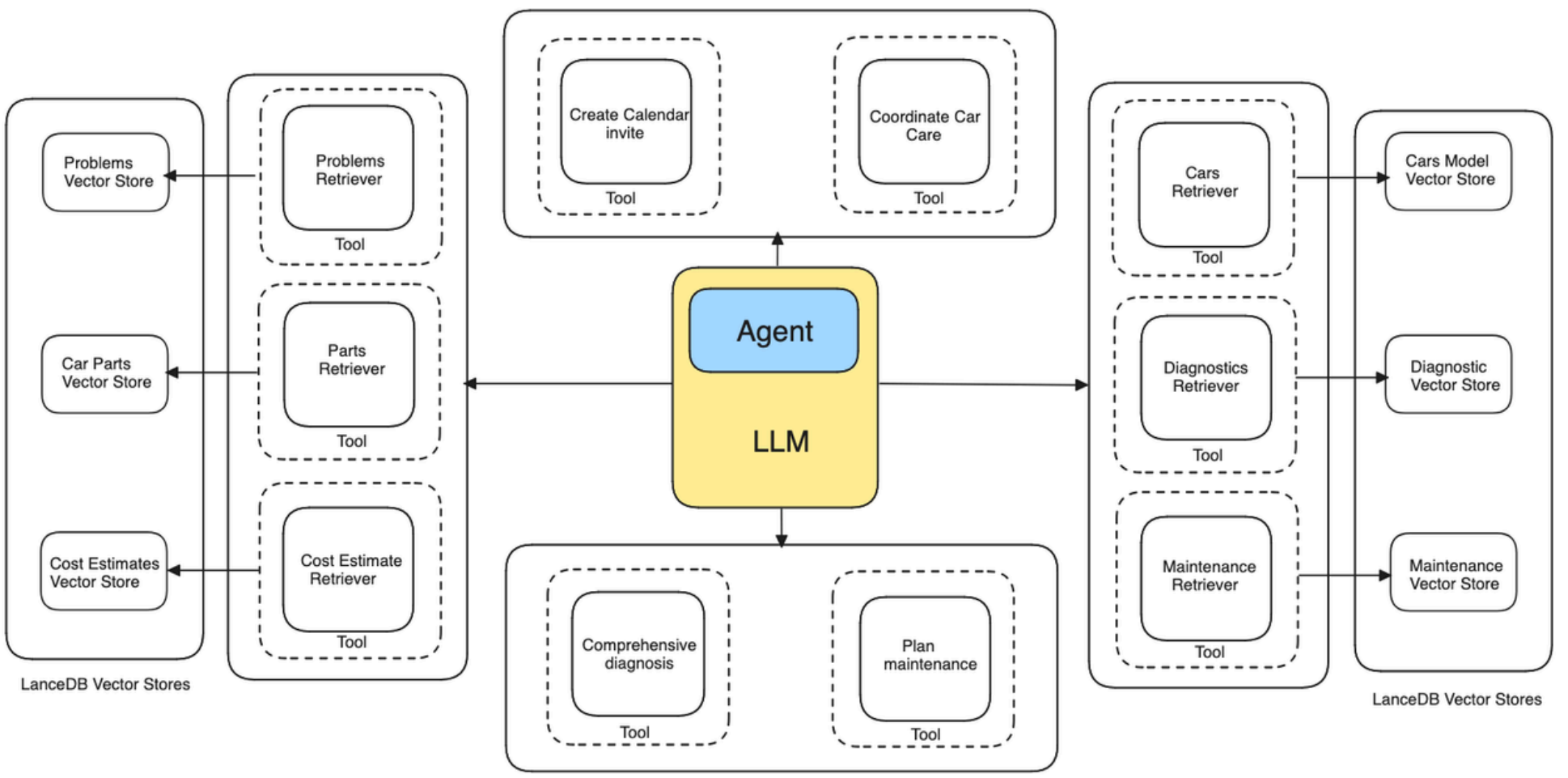
Multi-Agent Agentic RAG Systems

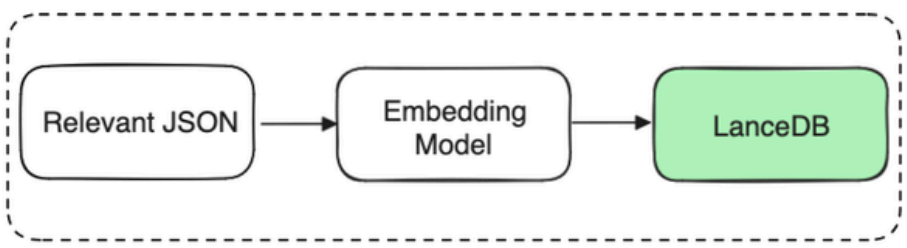




Multi-Agent Agentic RAG Systems

- A Multi-Agent Agentic RAG (Retrieval-Augmented Generation) system is an advanced AI architecture where multiple **autonomous agents collaborate** to enhance the efficiency, accuracy, and adaptability of Retrieval-Augmented Generation (RAG) models.
- These systems integrate agentic behaviors—meaning each agent can independently plan, reason, and execute tasks—into the RAG framework to improve its decision-making, retrieval, and response-generation capabilities.





All vector stores utilize LanceDB to store relevant embeddings for different types of data. For instance, the Car Parts Vector store holds embeddings for chunks created from the car_parts.json file, which contains information about various car parts. Similarly, other vector stores contain embeddings corresponding to their specific data.

Breaking the Terms

Multi-Agent System

- A multi-agent system consists of multiple AI agents that work together. Each agent has a specific role, expertise, or responsibility. These agents communicate, coordinate, and refine responses dynamically.

Agentic Behavior

- "Agentic" means the agents are autonomous, capable of independent reasoning, self-reflection, and adaptive learning. Unlike traditional AI models that passively retrieve or generate responses, agentic agents can evaluate responses, re-query databases, debate internally, and refine outputs.

Retrieval-Augmented Generation (RAG)

- RAG is a method that retrieves relevant documents from external sources (e.g., knowledge bases, vector databases, or APIs) and incorporates them into the generation process. This improves the accuracy and factual grounding of language models.
    -

Key Features and Advantages

- **Modularity**: Each agent operates independently, allowing for seamless addition or removal of agents based on system requirements.

- **Scalability**: Parallel processing by multiple agents enables the system to handle high query volumes efficiently.

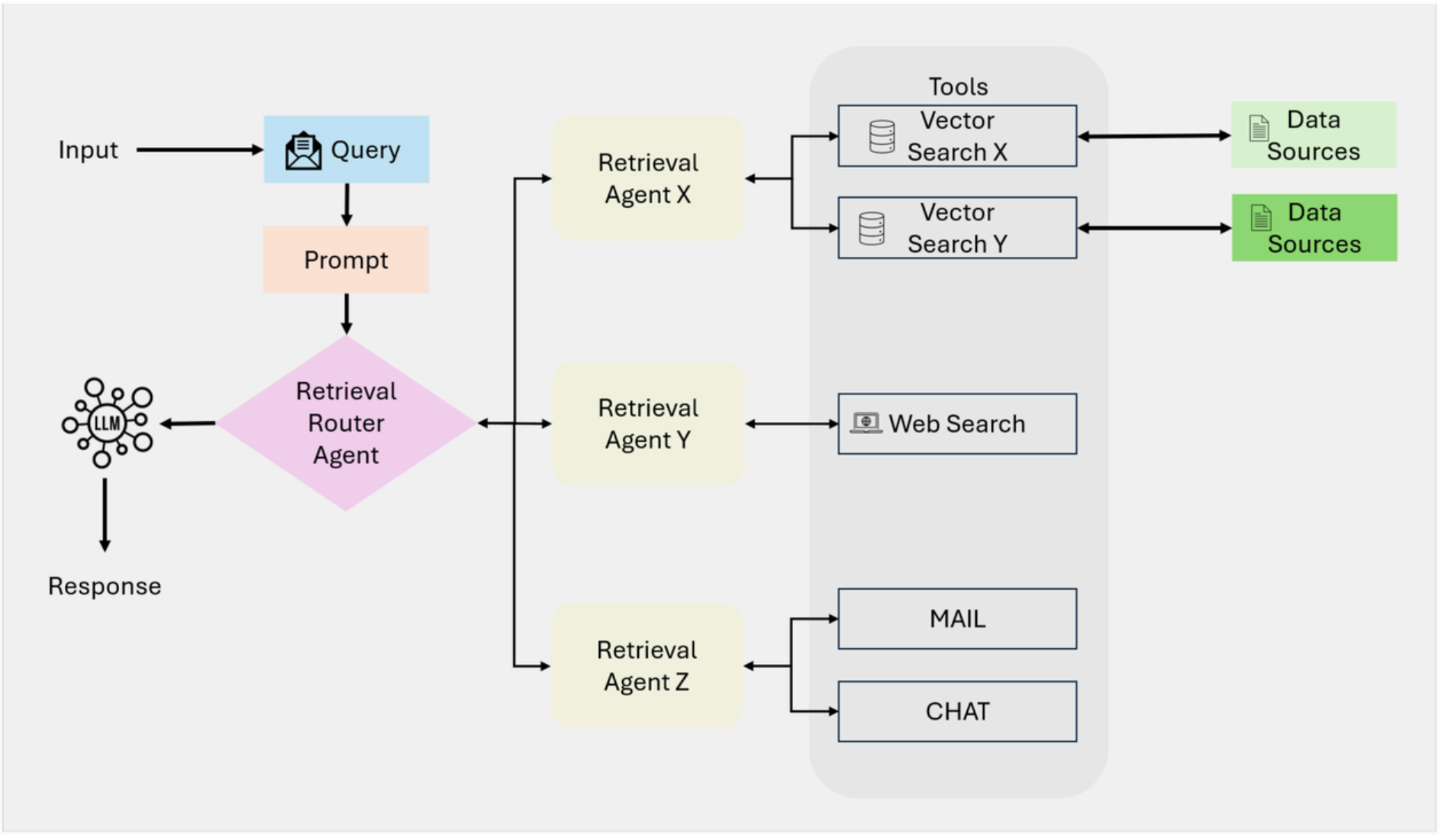
- **Task Specialization**: Each agent is optimized for a specific type of query or data source, improving accuracy and retrieval relevance.

- **Efficiency**: By distributing tasks across specialized agents, the system minimizes bottlenecks and enhances performance for complex workflows.

- **Versatility**: Suitable for applications spanning multiple domains, including research, analytics, decisionmaking, and customer support

Workflow

- **Query Submission**: The process begins with a user query, which is received by a coordinator agent or master retrieval agent. This agent acts as the central orchestrator, delegating the query to specialized retrieval agents based on the query's requirements.
- **Specialized Retrieval Agents**: The query is distributed among multiple retrieval agents, each focusing on a specific type of data source or task. Examples include:
    - **Agent 1**: Handles structured queries, such as interacting with SQL-based databases like PostgreSQL or MySQL.
    - **Agent 2**: Manages semantic searches for retrieving unstructured data from sources like PDFs, books, or internal records.
    - **Agent 3**: Focuses on retrieving real-time public information from web searches or APIs.
    - **Agent 4**: Specializes in recommendation systems, delivering context-aware suggestions based on user behavior or profiles.



- Tool Access and Data Retrieval: Each agent routes the query to the appropriate tools or data sources within its domain, such as:
    - **Vector Search**: For semantic relevance.
    - **Text-to-SQL**: For structured data.
    - **Web Search**: For real-time public information.
    - **APIs**: For accessing external services or proprietary systems.

The retrieval process is executed in parallel, allowing for efficient processing of diverse query types.

- **Data Integration and LLM Synthesis**: Once retrieval is complete, the data from all agents is passed to a Large Language Model (LLM). The LLM synthesizes the retrieved information into a coherent and contextually relevant response, integrating insights from multiple sources seamlessly.

- **Output Generation**: The system generates a comprehensive response, which is delivered back to the user in an actionable and concise format.

Challenges

- **Coordination Complexity**: Managing inter-agent communication and task delegation requires sophisticated orchestration mechanisms.

- **Computational Overhead**: Parallel processing of multiple agents can increase resource usage.

- **Data Integration**: Synthesizing outputs from diverse sources into a cohesive response is non-trivial and requires advanced LLM capabilities

Python Implementation

This implementation uses LangChain, FAISS, and OpenAI for intelligent multi-agent collaboration.

Key Components

- **RetrieverAgent** - Fetches relevant documents from a knowledge base.
- **SummarizerAgent** - Condenses retrieved information.
- **EvaluatorAgent** - Checks the quality and relevance of the summary.
- **ResponseGeneratorAgent** - Generates the final answer.

```
# Load Knowledge Base
loader = TextLoader("knowledge_base.txt")  # Replace with your file
documents = loader.load()

# Split text into manageable chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
docs = text_splitter.split_documents(documents)

# Create Vector Store for Retrieval
embeddings = OpenAIEmbeddings()
vector_store = FAISS.from_documents(docs, embeddings)
```

```python
# Define Multi-Agent System

class RetrieverAgent:
    """Fetches relevant documents from a vector store."""
    def __init__(self, vector_db):
        self.vector_db = vector_db

    def retrieve(self, query):
        retriever = self.vector_db.as_retriever()
        return retriever.get_relevant_documents(query)

class SummarizerAgent:
    """Summarizes retrieved documents for better clarity."""
    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4", temperature=0.3)

    def summarize(self, documents):
        combined_text = "\n".join([doc.page_content for doc in documents])
        summary_prompt = f"Summarize the following text:\n{combined_text}"
        return self.llm.predict(summary_prompt)

class EvaluatorAgent:
    """Evaluates the quality and relevance of retrieved documents."""
    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4", temperature=0.2)

    def evaluate(self, summary, query):
        eval_prompt = f"Evaluate if this summary properly answers '{query}':\n{summary}"
        return self.llm.predict(eval_prompt)

class ResponseGeneratorAgent:
    """Generates a final response based on evaluated information."""
    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4", temperature=0.7)

    def generate_response(self, evaluated_summary):
        response_prompt = f'Generate a detailed response based on the following evaluated summary:\n{evaluated_summary}'
        return self.llm.predict(response_prompt)

# Multi-Agent System Execution

class MultiAgentRAG:
    def __init__(self, vector_db):
        self.retriever = RetrieverAgent(vector_db)
        self.summarizer = SummarizerAgent()
        self.evaluator = EvaluatorAgent()
        self.generator = ResponseGeneratorAgent()

    def process_query(self, query):
        retrieved_docs = self.retriever.retrieve(query)
        summary = self.summarizer.summarize(retrieved_docs)
        evaluation = self.evaluator.evaluate(summary, query)
        final_response = self.generator.generate_response(evaluation)
        return final_response

# Instantiate Multi-Agent RAG System
multi_agent_rag = MultiAgentRAG(vector_store)

# User Query Execution
query = "What is the impact of AI on healthcare?"
response = multi_agent_rag.process_query(query)
print(response)
```

For complete code, follow my Github Repo:
https://github.com/prashant9501/Agentic-RAG

Share this post with others!

Repost

Data Science Manager with **15 years experience** in **AI** || Certified Generative AI Specialist || Experienced **AI Leader** & Coach for ML, DL, and NLP || PhD @ **IIT-Bombay**

Follow